REMARKS

In Response to Notice of Non-Compliant Amendment

Applicant herein submits a 37 CFR §1.131 declaration and a redacted copy of the Microsoft Confidential Specification, entitled "Functional Spec for XML Index on XML Blob Data Type Column in SQL Server Yukon" referenced in the declaration.

Respectfully submitted,

Date: August 15, 2008

/Joseph F. Oriti/
Joseph F. Oriti
Registration No. 47,835

Woodcock Washburn LLP
Cira Centre
2929 Arch Street, 12th Floor
Philadelphia, PA 19104-2891
Telephone: (215) 568-3100
Facsimile: (215) 568-3439

Functional Spec for XML Index on XML Blob Data Type Column in SQL Server Yukon

Author: {Shankar Pal (shankarp); Istvan Cseri (istvanc); Gideon Schaller (gideons); Oliver Seeliger (oliverse); Denis Altudov (daltudov); Denis Churin (denistc)}

Category: {XML Storage}

Date 6/3/2003 12:22 PM

Deleted: 4/29/2003 2:22:00 PM

Draft 0.4

Distribution: Microsoft Internal Distribution

Microsoft Confidential

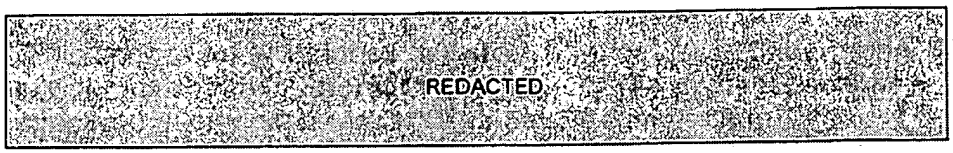


Table of Contents

1 Abstract and Context

In Yukon, a new scalar data type for XML data is being introduced. A relational table can have one or more XML columns that may be untyped or typed according to a specified XML schema. The XML values populating these columns are stored as blobs (varbinary (max)) for easier retrieval. For many purposes, the XML data type column behaves like an varbinary (max) column[1].

Queries are slow since the XML blob must be parsed each time a query is executed on it. If the result of parsing is saved, then queries can use it and run significantly faster. This necessitates shredding the XML data is into its basic components (*XML nodes*) and storing those in a separate *XML index*.

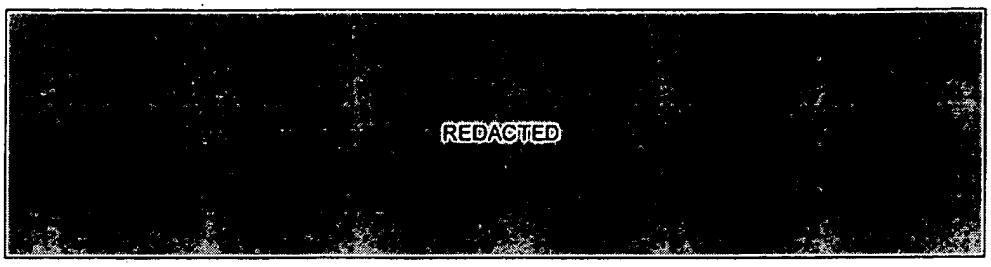
From the user's perspective, only the XML columns are visible, and the user creates the XML index on the XML column. This is not an ordinary index on a table – behind the scene, the engine creates structures that for convenience are called the *node table* in this document. The engine populates the node table from the XML blobs stored in the XML column in the spirit of a regular index. The engine components manage the correlation between the XML column and the underlying node table transparently, so that users continue to submit their queries on the XML column (instead of the node table). To prevent inadvertent use of the shredded XML form, users cannot bind to the node table directly for querying or manipulating the data stored therein.

In terms of implementation, the node table is created within the XML index creation DDL as an indexed view. Indexes on the node table are created by default.

[1] The CTypeInfo for the column has the value XVT_XML.

Deleted: 1 . Abstract and Context . 2¶
2 . Requirements, Background, Assumptions and Restrictions . 3¶
2.1 . Node Table . 3¶
2.2 . Indexes on Node Table . 4¶
3 . Description . 5¶
3.1 . XML Index Syntax . 5¶
4 . Optional XML Indexes . 6¶
4.1 . Basic XML Index . 6¶
4.2 . XML Index Extension . 7¶
4.3 . XML Index Extension — Path Index . 8¶
4.3.1 . Effect of Hierarchical Index . 8¶
4.4 . XML Index Extension — Value Index . 9¶
4.5 . XML Index Extension — CHILD Index . 9¶
4.5.1 . Effect of Creating CHILD Index . 9¶
5 . XML Index Extension — Node Index . 10¶
6 . Combining Full-text Index with XML Index . 10¶
6.1 . XML Index Extensibility Strategy . 10¶
6.2 . Drop XML Index . 11¶
6.3 . Drop XML Column and Drop Table Containing XML Columns . 11¶
6.4 . Altering XML Indexes . 11¶
6.5 . Miscellaneous Operations on XML Index . 12¶
6.6 . Transaction Isolation Levels . 13¶
7 . Index Behavior in other cases . 13¶
7.1 . Replication . 13¶
7.2 . DBCC . 13¶
7.3 . BCP . 14¶
7.4 . Partitioning . 15¶
8 . System-defined Table-valued Function . 15¶
9 . Implementation of XML Index . 16¶
9.1 . Indexed View Implementation . 16¶
9.1.1 . View Definition . 16¶
9.1.2 . Clustered Index Definition . 17¶
9.1.3 . Secondary Index Definitions . 18¶
9.1.4 . Modifying XML Instances . 18¶
9.1.5 . Using SET Options . 18¶
10 . Scenarios and Examples . 19¶
11 . Other Areas of Impact . 19¶
12 . Considerations for Other Features, Components, and Products . 19¶
13 . Known Work Outstanding . 20¶
14 . Document Change History . 20¶

Formatted: Do not check spelling or grammar



2.1 Node Table

Storage for XML columns X and Y in a user table R and its underlying node tables is as follows:

R

PK	XML X	XML Y
201	<a>...	...
301	<a>...	NULL

NTX

PK	XID	PID	NID	TID	VALUE	LVALUE
	OrdPath	*OrdPath*	*Integer*	*Integer*	*Variant*	*Varbinary<max>*
201	1	NULL		...	...	...
201	1.1	1		...	...	...
201	1.3	1		...	...	...
301	1	NULL		...	...	...
301	3	NULL		...	...	...

NTY

PK	XID	PID	NID	TID	VALUE	LVALUE
	OrdPath	*OrdPath*	*Integer*	*Integer*	*Variant*	*Varbinary<max>*
201	1	NULL		...	...	...
201	1.1	1		...	...	...
201	1.3	1		...	...	...
201	1.3.5	1.3		...	...	...

In the node table, XID is an identifier (*OrdPath*) for the XML node represented by the row. It captures hierarchical relationship and the order among the nodes in the XML data (called *document order*). The XID of the node's parent is stored in the computed PID column. PID is a prefix of the node's XID and is an OrdPath itself.

NID is a token value for the XML node name, while TID is a token value for the type of the XML node. The VALUE column stores the value, if one exists, of the node up to a pre-determined length (128 bytes). Longer values are stored in the LVALUE column with the first 128 bytes stored in the VALUE column as well.

The clustered, primary key of the node table is composed of the primary key of the user table and the XID column; hence, the primary key of the user table cannot contain more than 15 columns. If the clustered, primary key in the user table contains 16 columns, then no XML column can be added to the table.

2.2 Indexes on Node Table

Indexes on the node table are the following:

- Clustered index on (PK, XID), which causes clustering in depth-first order of the XML hierarchy
- NAME INDEX: Secondary index on (NID, TID, PID, PK, and XID).
- VALUE INDEX: Secondary index on (VALUE, NID, TID, PK, and XID).

The NAME index is created automatically since it is useful for almost all queries on the XML data. An example is /Customer/Order, in which this index is looked up with the token values #Customer and #Order to determine the corresponding XML nodes.

The VALUE index is useful for value-based queries, and is created at the same time as the node table. Optional value indexes will be considered for future releases.

3 Description

3.1 XML Index Syntax

The syntax for XML indexes is the standard one for creating an index except that only a single, XML column is involved:

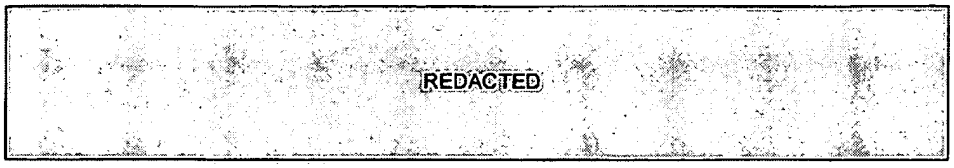
```
CREATE XML INDEX index_name ON table (Column)
      [WITH (<index_option> [,...n ]) ]

<index_option> ::=
        PAD_INDEX  = {ON | OFF}
      | FILLFACTOR = fillfactor
      | SORT_IN_TEMPDB  = {ON | OFF}
      | STATISTICS_NORECOMPUTE  = {ON | OFF}
      | DROP_EXISTING = {ON | OFF}
      | ALLOW_ROW_LOCKS= {ON | OFF}
      | ALLOW_PAGE_LOCKS={ON | OFF}
      | MAXDOP=number_of_processors
```

This creates the necessary node table structure and secondary indexes on it, and populates those structures from the values in the XML column. All indexing options are available, except IGNORE_DUP_KEY and ONLINE, which are always OFF. For example, if the DROP_EXISTING option is specified, then the existing XML index is dropped and a new one created in its place.

The default values for the options are as follows:

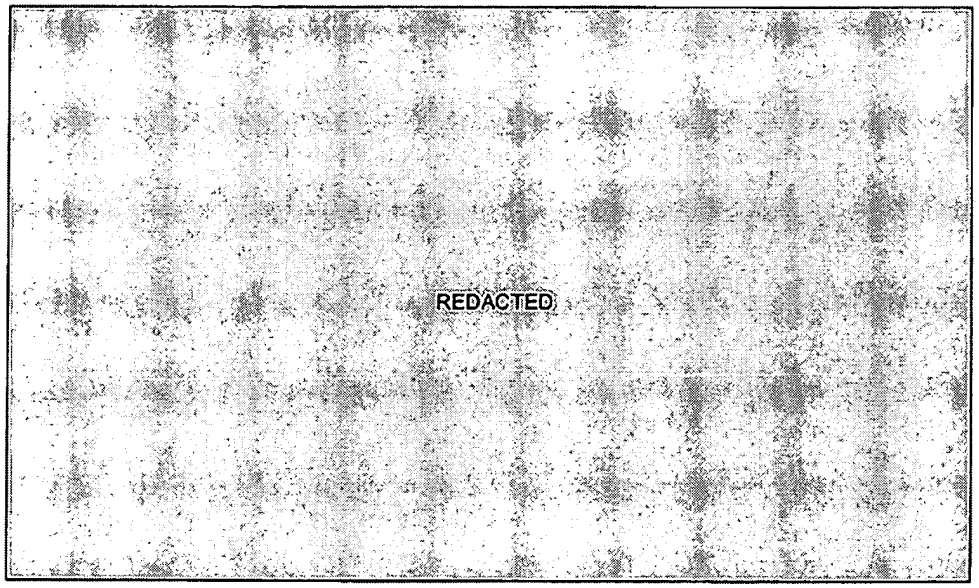
```
PAD_INDEX = OFF
FILLFACTOR = 0
SORT_IN_TEMPDB = OFF
STATISTICS_NORECOMPUTE  = OFF
DROP_EXISTING = OFF
ALLOW_ROW_LOCKS= ON
ALLOW_PAGE_LOCKS= ON
MAXDOP = 0
```

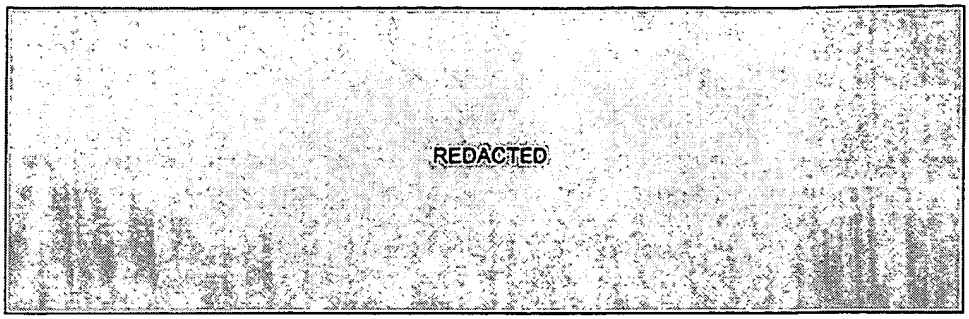
An XML index creation is permitted if the user table contains a clustered, primary key. Otherwise, XML index creation fails. If an XML index exists on any XML column in a table, the clustered primary key of the table cannot be modified. Users have to drop all XML indexes on the table before modifying the clustered, primary key.



Structures necessary to support the indexing behavior (e.g. indexed view) is created as part of this DDL. These structures are populated with the values from the XML column. Updates occurring on the XML column are propagated to these XML index structures as well. Since XML data type instances are incomparable, no statistics can be created on an XML column.

As a design philosophy, actions on the XML index should be propagated to the underlying storage (e.g. indexed view) for user-friendly behavior. However, it is not always desirable or feasible to do so. The following subsections provide the feature description of XML index.





3.2 Optional XML Indexes

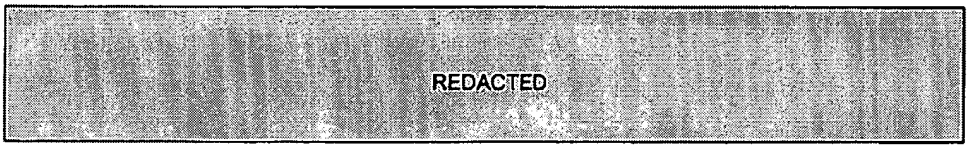
Customers predominantly want the ability to specify the parts of XML instances that should be indexed. That is, they want greater control over the granularity of to be indexed. This saves space compared to indexing the full XML instances. Consequently, insertions and updates to the XML column are faster.

We can provide more control in the following ways:

1. Allow users to create the index on tags. Only the tag or the subtree under the tag (*partial shredding*) may be indexed. By default, the XML index then creates the node table only. In a variation, the sub-tree can be stored as a blob for easier retrieval.
2. Allow users to create an index on values for value-based searches such as //section[@* = 6].
3. Allow users to specify the paths on which tags and values are to be indexed. There paths may be relative or absolute. Alternatively, we can allow users to specify the paths which should be excluded from indexing.
4. Allow users to index specified paths.
5. Allow full-text index to be built on the XML index.
6. Allow users to index children of nodes. This is useful for navigational access.

These options – except partial shredding and possibly full-text index on XML index – check for satisfiability of the search condition on the instances in the XML column. The subtrees under the nodes that satisfy the search condition are not retrieved. Entire XML instances may be returned. To extract the subtrees from under the nodes satisfying the search condition, an outer query must be executed to do so.

Partial shredding can return the subtree but serializing it out from the index.



3.4 Drop XML Index

The syntax for dropping an XML index is the regular DROP INDEX command except that an XML index is indicated and an XML index name is specified.

```
DROP INDEX index_name ON table  [ ,...n ]
```

This results in dropping the XML index from meta-data and all associated storage. For example, if the underlying implementation is indexed view, then the indexed view and its indexes are dropped as well.

Whenever an XML index is dropped, the QNAME entries occurring in the XML index storage are deleted from meta-data storage.

The DROP INDEX statement does not support the deprecated syntax (DROP INDEX Table.Index).

3.5 Drop XML Column and Drop Table Containing XML Columns

An XML column is dropped using the regular ALTER TABLE command. This causes the column to be deleted from the table following the usual engine logic. If an XML index exists on the column, then the column cannot be dropped but returns an error. This behavior is consistent with columns of non-XML types.

When a table containing one or more XML columns is dropped, all XML indexes defined on those XML columns are dropped as well. This causes the supporting structures of the XML indexes to be dropped as well.

3.6 Altering XML Indexes

Altering an XML index is allowed for modifying B-tree options (e.g. re-building the indexes). For XML index, the appropriate indication is supplied in the DDL below:

```
ALTER {INDEX  {index_name | ALL} ON {table | view}
      { REBUILD [WITH ( <rebuild_index_option> [ ,...n] ) ]
| DISABLE
| REORGANIZE [ WITH (LOB_COMPACTION={ON | OFF}) ]
| SET (<set_index_option> [ ,...n] )
}

<rebuild_index_option[2]> ::=
     PAD_INDEX  = {ON | OFF}
   | FILLFACTOR = fillfactor
   | SORT_IN_TEMPDB = {ON | OFF}
   | STATISTICS_NORECOMPUTE  = {ON | OFF}
   | ALLOW_ROW_LOCKS= {ON | OFF}
   | ALLOW_PAGE_LOCKS={ON | OFF}
   | MAXDOP=number_of_processors

<set_index_option> ::=
   ALLOW_ROW_LOCKS= {ON | OFF}
```

[2] DROP_EXISTING is not allowed in the ALTER INDEX syntax.

```
| ALLOW_PAGE_LOCKS={ON | OFF}
| STATISTICS_NORECOMPUTE  = {ON | OFF}
```

In Yukon, an XML index can be created on an XML column in a table but not in a view. Hence, a view name specified for an XML index in the ALTER INDEX statement results in an error.

The rebuild options ONLINE and IGNORE_DUP_KEY, and the set option IGNORE_DUP_KEY are not valid for XML index.

Modification of the PK constraint in the user table is not automatically propagated to XML indexes; rather, the burden is on the user to drop all XML indexes on the table and to re-create them. If the user attempts to change the PK constraint while XML indexes exist on the table, a meta-data error is generated.

Whenever altering an XML index causes re-building the index, QNAME cleanup occurs in meta-data. The QNAME entries in meta-data that no longer occur in the XML index structures are removed from meta-data storage.

The DISABLE option is supported on an XML index. If this option is set, the index is no longer used in query plans and not maintained.

If the option ALL is specified, it applies to both non-XML and XML indexes. Other options may be specified that are not valid for both types of indexes (e.g. ONLINE=ON option for an XML index, or IGNORE_DUP_KEY=ON for a non-unique index or an XML index). In such cases, the entire statement fails.

3.7 Miscellaneous Operations on XML Index

Although an XML index is created in the same namespace as non-XML indexes, some operations on XML indexes may not work. Examples of such operations are:

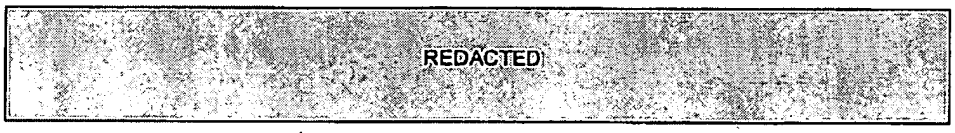
- Sp_helpIndex
  Such stored procedures are being deprecated. Instead, the XML index information will be found in catalog views.

Meta-data functions – INDEX_COL, INDEXKEY_PROPERTY – which accept index id as arguments will work for XML index.

DBCC CHECKTABLE (table, index-id) will look for a regular and XML index on the table and check the XML index as a regular table with non-clustered indexes.

3.8 Transaction Isolation Levels

All transaction isolations all supported on XML data type. The behavior is similar to other first-class relational data types.



4 Index Behavior in other cases

4.1 Replication

If the XML index is created on an XML column, the index is automatically replicated. This creates the underlying storage (i.e. indexed view) at the other end of replication. The XML blobs are written into the log for transactional replication, just like text columns. Thus, text and XML columns are treated the same way for replication.

Data in the underlying storage for XML index (i.e. the indexed view) are *not* replicated. Rather, at the target end, the XML blobs are pieced together from the log the same way as text data, and assigned to the corresponding XML column. The XML index mechanism ensures that the update is propagated to the underlying storage (indexed view).

If the XML data type column is constrained by a schema, the constraining XML schema must be available at the subscriber for replication of the XML column.

If XML index is implemented as indexed view, replication of the indexed view is not allowed directly but results in an error. Allowing it yields significant complexity without adding much value.

4.2 DBCC

DBCC commands involving an XML index are propagated to the underlying storage (indexed view). This preserves users' perception of an XML index being analogous to a secondary index on any other SQL type.

Currently, DBCC commands on a table T do not perform the same checks on an indexed view V on T; the calls must be made separately on V. However, propagating the DBCC checks to indexed views requires SE and QP enhancements.

DBCC Function	Action on XML Blob column
CHECKCATALOG	This is a meta-data consistency check. There is no work for XML index created as indexed view
CHECKCONSTRAINTS	Performed by existing DBCC functionality
CHECKDB	No logical XML checks are performed on the XML blob. This assumes that physical checks are adequate to ensure the indexed view is in sync with the XML blobs. If there is a discrepancy, the XML index can be re-built.
CHECKFILEGROUP	This DBCC function behaves the same way as CHECKDB but is scoped to the specified file group.
CHECKTABLE	1. Physical checks are assumed to be adequate, and no well-formedness or validation of XML blobs is needed. 2. CHECKTABLE is not propagated to the underlying storage

	(indexed view) whether or not NOINDEX is specified. Instead, users must directly make the DBCC call on the indexed view (which should be allowed although the index view is non-bindable) and provides the current behavior. 3. If REBUILD option is specified, then the XML index is regenerated. This results in QNAME cleanup as well.
DBREINDEX (ALTER XML INDEX ... REBUILD)	1. The XML index is re-built. This requires regeneration of the underlying storage. 2. QNAME entries are cleaned up.
INDEXDEFRAG (ALTER XML INDEX ... REORGANIZE)	Similar issues as DBREINDEX
SHOW_STATISTICS	1. Statistics on an XML column is not allowed, so that this command returns an error. 2. Statistics cannot be created, updated and viewed on columns in the underlying storage (indexed view). For indexed view implementation, the indexed view is non-bindable[3].
SHOWCONTIG	Same issues as SHOW_STATISTICS
UPDATEUSAGE	Same issues as SHOW_STATISTICS

The actions required for DBCC are dependencies on the SE team.

4.3 BCP

BCP out of the user table copies the XML blobs; neither the XML index definition nor the content of the underlying storage is output. Users cannot BCP out the underlying indexed view since it is non-bindable.

The XML blob can be loaded into a similar column at the destination database. If an XML index definition exists at the target, then it is populated. Users may expect this behavior since indexes on non-XML columns are handled the same way.

BCP in requires well-formedness check for XML as well as validation if an XML schema is attached to the target XML column.

If the XML data type column is constrained by a schema, the constraining XML schema must be available at the target for BCP in to succeed.

4.4 Partitioning

XML data type instances are internally stored as LOB in the user table. The partitioning or filegroup specification on the user table applies to it. For example, if LOBs are to be stored in a different partition or filegroup than the rest of the user table, then XML blobs will follow the same pattern.

The XML index (internal table) is in the same partition or filegroup as the user table it is defined in (and not collocated with the LOB partition/filegroup in the user table if the LOB

[3] "Non-bindable" means that a user cannot bind to the content or the schema of the indexed view using T-SQL or DDL.

partition/filegroup is different from that of the user table). This yields faster query performance at the expense of data modification speed.

This requires that

- The partitioning scheme defined on the user table is applied to the underlying storage for XML index.
- The clustered, primary key is copied from the user table into the underlying storage for XML index. While this can be relaxed a little, this restriction turns out to be the most straightforward and robust one.

When the partitioning scheme is modified (e.g. to specify NEXT_USED), the modification is propagated to the underlying storage.

Non-clustered indexes on the internal table of the XML index are in the same partition/filegroup as the internal table.

When the partitioning scheme of the user table is changed to a new one, if the partitioning columns are different, the XML index must be re-generated to include the partitioning columns. Hence, such partitioning scheme changes result in error as long as one or more XML indexes exist on the table. The user must explicitly (using DROP INDEX statement) drop all XML indexes on the table, change the partitioning schema, and re-build the indexes.

No XML indexes can exist on the user table when the clustered index on the user table is re-created with DROP_EXISTING = ON and a new partitioning scheme. On an XML index exist, then the re-creation fails.

5 System-defined Table-valued Function

The table-valued function is created as follows:

```
CREATE FUNCTION SHRED_XML (
      @xmlcol  XML)                    -- XML column on which to execute
RETURNS @ShreddedRows TABLE (
      XID          OrdPath,            -- node id
      NID          int,                -- the name ID value
      TID          int,                -- the type ID value
      VALUE        sql_variant,        -- (small) value column
      LVALUE       nvarchar(max),      -- large value column
      HID          int                 -- tokenized path
      ) AS
BEGIN
      DECLARE      @Temp TABLE (
                   XID OrdPath, NID int, TID int, VALUE sql_variant,
                   LVALUE nvarchar(max), ...)

      INSERT       INTO @Temp
      SELECT       *
      FROM         Rowset-generated-by-shredding-XML-BLOB-@xmlcol-with-
                   path-expression-and-max-level-applied

      INSERT       INTO @ShreddedRows
```

```
        SELECT      XID, NID, TID, VALUE, LVALUE, HID FROM @Temp
        RETURN
END
```

The XML column is passed in as an argument as required.

6 Implementation of XML Index

When a user issues the CREATE INDEX statement on an XML data type column, a meta-data object is created for the index. In addition, auxiliary structures are created automatically within the same DDL (i.e. not explicitly by the user) to implement the node table as an indexed view.

The following subsections describe the behind-the-scenes activities that go on to implement the node table. The XML subsystem uses the table-valued function of Section 5 to create the indexed view.

Deleted: 7

6.1 Indexed View Implementation

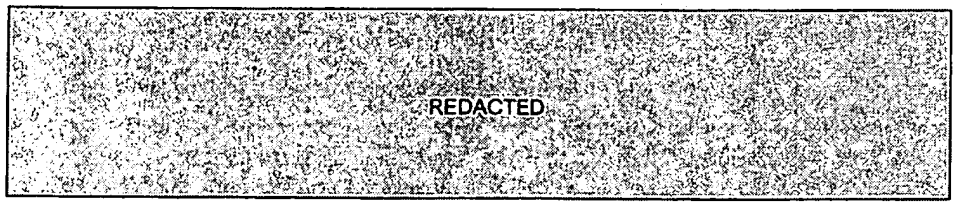
6.1.1 View Definition

The indexed view is created as

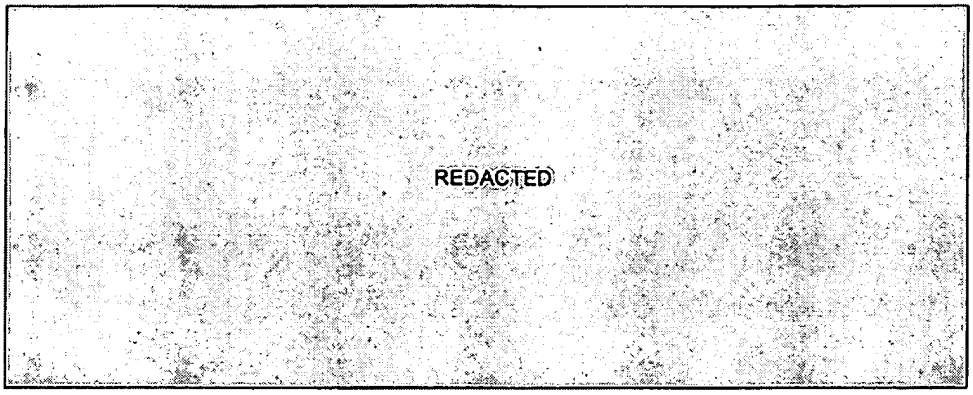
```
CREATE VIEW index-name AS
      SELECT      T.CK, NT.*
      FROM        table T
                  APPLY SHRED_XML (X) NT
```

Where CK is the non-null, clustering key in the user table on which the view is being created. When XML values in the user table are updated, the changes are propagated to the indexed view. In principle, only the incremental difference needs to be propagated. However, owing to the occurrence of the table-valued function, the computation of the difference is hard. Instead, the rows in the node table corresponding to the old XML values are deleted, and new ones for the updated XML values are inserted. This generates a significant number of log records for the node table.

The indexed view is non-updatable but visible to the user in the catalog. To avoid any user dependency on its contents, the view is made non-bindable in meta-data, so that user queries against the indexed view are failed by QP.



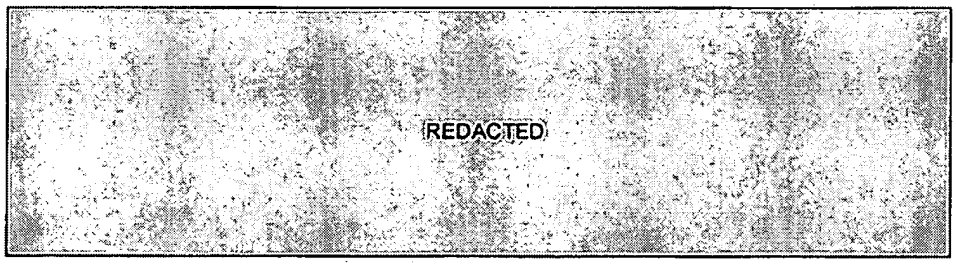
Since the indexed view is visible, users can freely create (CREATE STATISTICS), view (DBCC SHOW_STATISTICS) and update statistics (UPDATE STATISTICS) on the individual columns of the indexed view.



6.1.2 Clustered Index Definition

Once this view has been created, the first clustered index on it is defined as

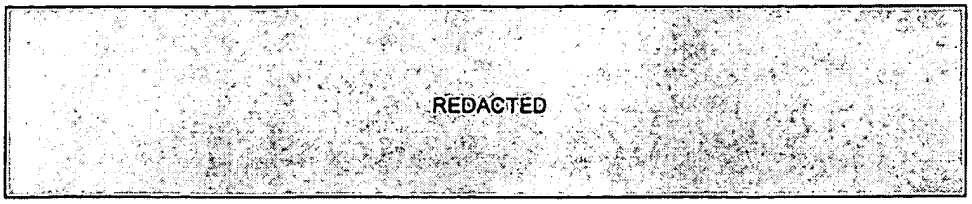
```
CREATE UNIQUE CLUSTERED INDEX internal-index-name
    ON index-name (CK, XID)
    [WITH <index_option> [,...n ] ]
    [ON {partition_scheme_name (column_name [1,...n]) | filegroup }]
```



6.1.3 Secondary Index Definitions

Thereafter, the secondary indexes are created.

Name index	`CREATE INDEX internal-index-name` `ON index-name (NID, TID)`
Value index	`CREATE INDEX internal-index-name` `ON index-name (VALUE, NID, TID)`



6.1.4 Modifying XML Instances

In principle, the delta change can be propagated by the QP to the indexed view. This, however, is quite difficult to achieve since a table-valued function is involved in the view definition.

A much simpler strategy – the one adopted for the Yukon release – is for QP to replace the existing XML instance with the new one. This causes the corresponding, old rows in the node table to be dropped and the new ones inserted. This approach suffers from the following drawbacks:

- A log record is generated for each row deleted from and inserted into the indexed view.
- Both the old and the new states of the XML blob are logged. That is, the log records contain more than just the difference between the two values. Consequently, the log is bigger. This impacts scenarios such as replication.

6.1.5 Using SET Options

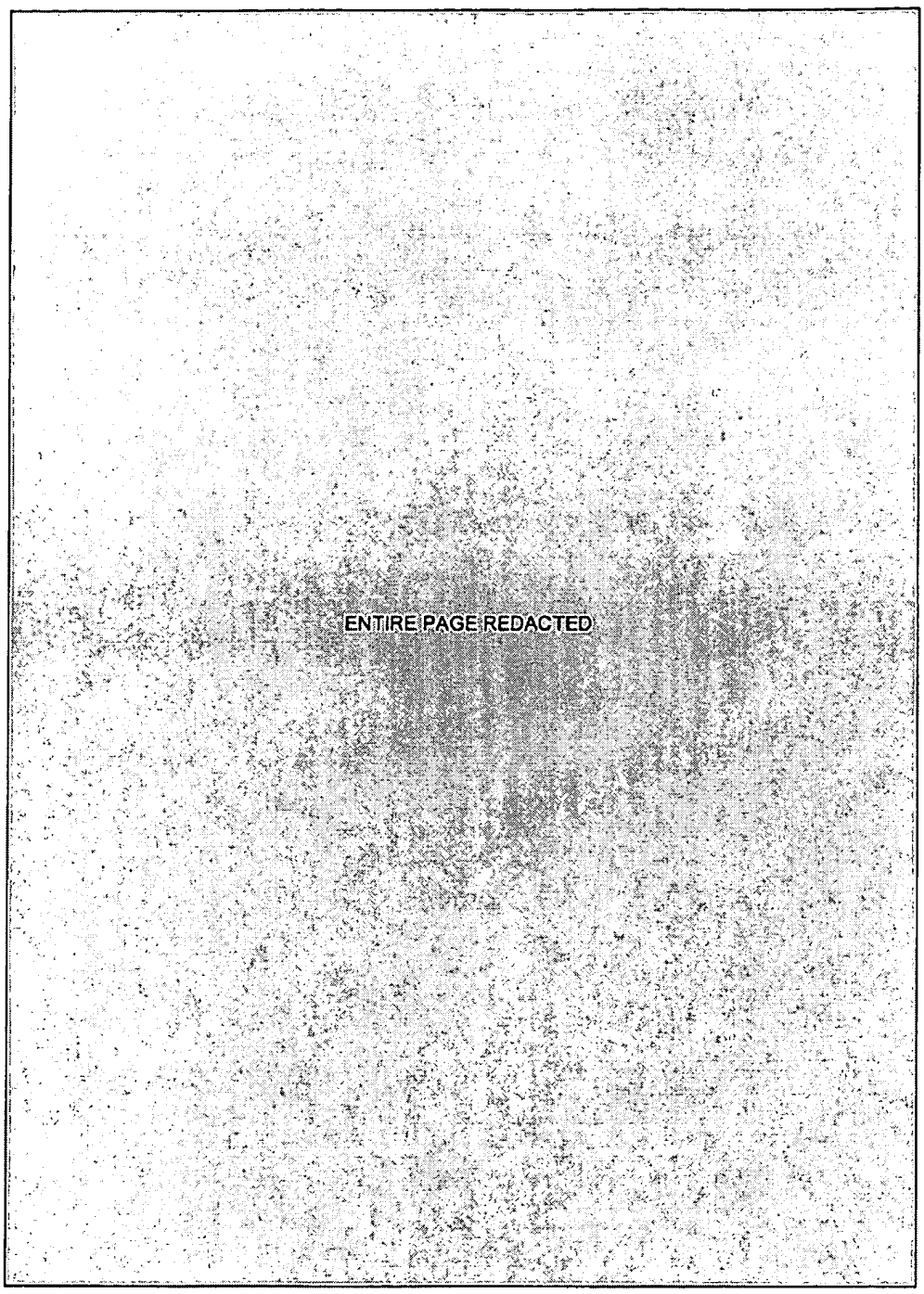
Indexed view requires SQL options for consistent results. These are shown in the table below.

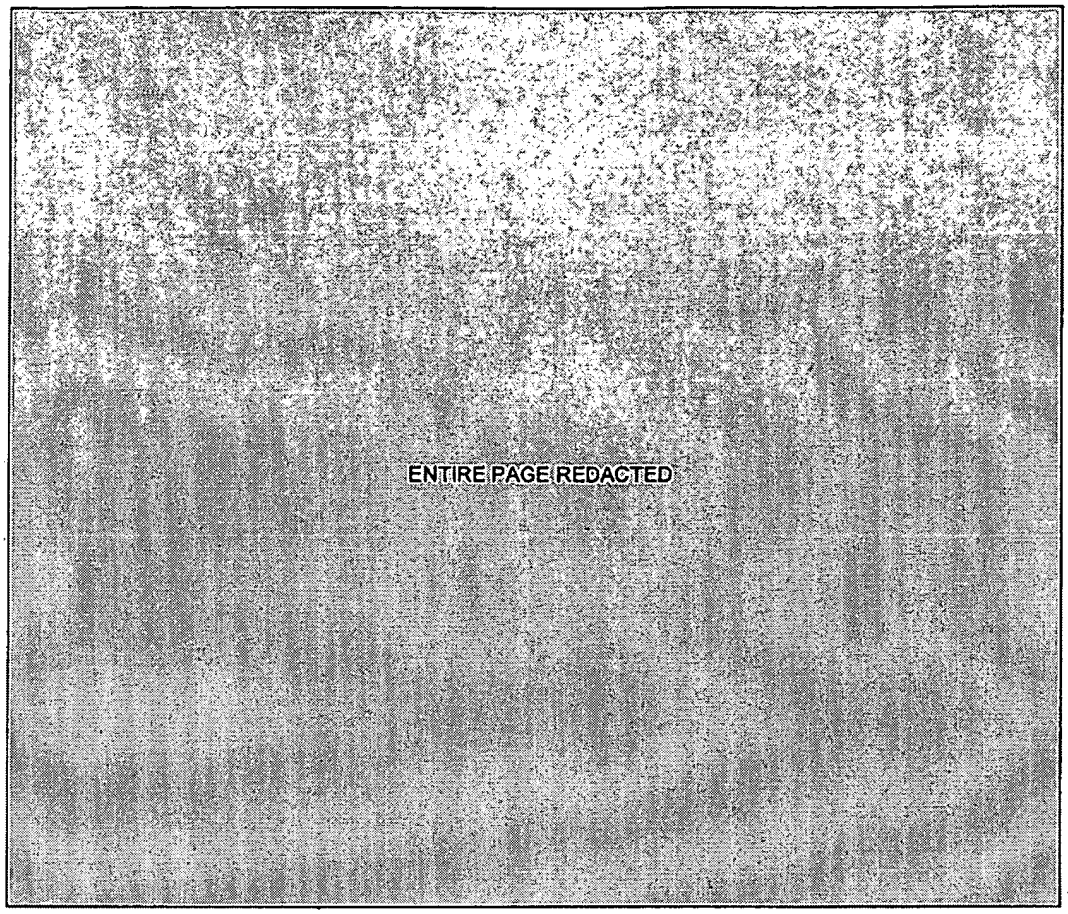
SET Options	Required Value for Indexed View	Default Server Value	OLE DB & ODBC Value	DB LIB Value	HTTP Value
ANSI_NULLS	ON	OFF	ON	OFF	ON
ANSI_PADDING	ON	ON	ON	OFF	ON
ANSI_WARNING	ON	OFF	ON	OFF	ON
ARITHABORT	ON	OFF	OFF	OFF	ON
CONCAT_NULL_YIELDS_NULL	ON	OFF	ON	OFF	ON
NUMERIC_ROUNDABORT	OFF	OFF	OFF	OFF	OFF
QUOTED_IDENTIFIER	ON	OFF	ON	OFF	ON

The SQL options must be set to the values shown in the **Required Value** column whenever these conditions occur:

- The XML index is created, since an underlying indexed view may be created.
- There is an INSERT, UPDATE, or DELETE operation performed on any XML column participating in an XML index, or an update of the PK column values in the user table.

Deleted: y


ENTIRE PAGE REDACTED


ENTIRE PAGE REDACTED